A Framework for Dynamic Parameterized Dictionary Matching*

Arnab Ganguly[1], Wing-Kai Hon[2], and Rahul Shah[3]

1 **School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, USA**
agangu4@lsu.edu,rahul@csc.lsu.edu
2 **Department of Computer Science, National Tsing Hua University, Hsinchu City, Taiwan**
wkhon@cs.nthu.edu.tw
3 **School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, USA; and National Science Foundation, Arlington, USA**
rahul@nsf.gov

Abstract

Two equal-length strings S and S' are a parameterized-match (p-match) iff there exists a one-to-one function that renames the characters in S to those in S'. Let $\mathcal{P}$ be a collection of d patterns of total length n characters that are chosen from an alphabet Σ of cardinality σ. The task is to index $\mathcal{P}$ such that we can support the following operations:

- $\mathsf{search}(T)$: given a text T, report all occurrences $\langle j, P_i \rangle$ such that there exists a pattern $P_i \in \mathcal{P}$ that is a p-match with the substring $T[j, j + |P_i| - 1]$.
- $\mathsf{insert}(P_i)/\mathsf{delete}(P_i)$: modify the index when a pattern P_i is inserted/deleted.

We present a linear-space index that occupies $\mathcal{O}(n \log n)$ bits and supports (i) $\mathsf{search}(T)$ in worst-case $\mathcal{O}(|T| \log^2 n + occ)$ time, where occ is the number of occurrences reported, and (ii) $\mathsf{insert}(P_i)$ and $\mathsf{delete}(P_i)$ in amortized $\mathcal{O}(|P_i| \operatorname{polylog}(n))$ time. Then, we present a succinct index that occupies $(1 + o(1))n \log \sigma + \mathcal{O}(d \log n)$ bits and supports (i) $\mathsf{search}(T)$ in worst-case $\mathcal{O}(|T| \log^2 n + occ)$ time, and (ii) $\mathsf{insert}(P_i)$ and $\mathsf{delete}(P_i)$ in amortized $\mathcal{O}(|P_i| \operatorname{polylog}(n))$ time. We also present results related to the semi-dynamic variant of the problem, where deletion is not allowed.

1998 ACM Subject Classification F.2.2 Pattern Matching

Keywords and phrases Parameterized Dictionary Indexing, Generalized Suffix Tree, Succinct Data Structures, Sparsification

Digital Object Identifier 10.4230/LIPIcs.SWAT.2016.10

1 Introduction

Designing succinct data structures for the classical pattern matching problem of finding all occurrences of a pattern P in a fixed text T can be traced back to the seminal work of Grossi and Vitter [16], Ferragina and Manzini [13], and Sadakane [29]. This established an active research area of designing succinct data structures. The focus was now on either improving these initial breakthroughs (see [26] for a comprehensive survey), or designing succinct data

* The work of Arnab Ganguly was supported by National Science Foundation Grants CCF–1017623 and CCF–1218904. The work of Wing-Kai Hon was supported by National Science Council Grants 102-2221-E-007-068-MY3 and 105-2918-I-007-006.

structures for other variants [6, 8, 14, 25, 30]. *Dictionary Matching*, a typical example of these variants, is defined as follows. Let $\mathcal{P}$ be a collection of d patterns $\{P_1, P_2, \ldots, P_d\}$ of total length n characters which are chosen from a totally-ordered alphabet Σ of cardinality σ. Given a text T, also over Σ, the task is to report all positions j such that at least one of the patterns $P_i \in \mathcal{P}$ exactly matches an equal-length substring of T that starts at j. Typically, the patterns which occur at j are also reported. In the *Dictionary Indexing* problem, the patterns are provided upfront (and remain fixed) and the text comes as a query. The classical solution for this problem is the Aho-Corasick (AC) automaton [1] which occupies $\mathcal{O}(m \log m)$ bits of space, where $m \leq n+1$ is the number of states in the automaton, and finds all *occ* occurrences in optimal time $\mathcal{O}(|T| + occ)$. To the best of our knowledge, the first succinct index for this problem is by Hon et al. [17]. Later, Belazzougui [6] presented an $m \log \sigma + \mathcal{O}(m + d \log(n/d))$ bit index with optimal $\mathcal{O}(|T| + occ)$ query time.

Arguably, the most natural variant of the dictionary indexing problem is the *Dynamic Dictionary Indexing* problem in which we are allowed to insert a new pattern or delete an existing one. The challenge is to modify the index under such updates without having to rebuild it from scratch. Of course, the index should still be able to answer text queries in a reasonably efficient time. The first non-trivial solution was provided by Amir et al. [3]. As opposed to the AC-automaton, their technique comprised of the generalized suffix tree GST of all the patterns. Later, this was improved by Amir et al. [4] and again by Alstrup et al. [2]. Each of these indexes occupies $\mathcal{O}(n \log n)$ bits of space. The natural question to ask is "Does there exist a succinct index for dynamic dictionary matching?". Chan et al. [9] answered this by presenting an $\mathcal{O}(n\sigma)$-bit index which is (nearly) succinct only for $\sigma = \mathcal{O}(1)$. Moreover, their query time suffers from an $\mathcal{O}(\log^2 n)$ multiplicative slowdown (compared to the AC-automaton) due to the use of complicated dynamic versions of FM-Index [13] and Compressed Suffix Tree [16] as the underlying main ingredient. Updating the index for a pattern P_i required $\mathcal{O}(|P_i| \log^2 n)$ time. Hon et al. [18] improved this to a more space efficient $(1 + o(1))n \log \sigma + \mathcal{O}(d \log n)$ bit index with a faster $\mathcal{O}(|T| \log n + occ)$ query time and $\mathcal{O}(|P_i| \log \sigma + \log n)$ update time. Recently, Feigenblat et al. [12] improved the query time to $\mathcal{O}(|T|(\log \log n)^2 + occ)$ for $\sigma = \mathcal{O}(\text{polylog}(n))$.

Parameterized Pattern Matching has received significant attention (see [22] for a survey) since its inception by Baker [5]. The alphabet Σ is partitioned into two disjoint sets: Σ_s containing static-characters (s-characters) and Σ_p containing parameterized characters (p-characters). Two strings S and S', both over Σ, are a parameterized-match (p-match) iff $|S| = |S'|$, and there is a one-to-one function f such that $S[i] = f(S'[i])$. For any s-character $c \in \Sigma_s$, we have $f(c) = c$. Thus, for $\Sigma_s = \{A, B, C\}$ and $\Sigma_p = \{w, x, y, z\}$, the strings $AxBxCy$ and $AzBzCx$ are p-match, but $AxBxCy$ and $AzBwCx$ are not. We consider the *Parameterized Dictionary Matching* problem which was introduced by Idury and Schäffer [19]. This is similar to the standard dictionary problem, just that the alphabet Σ is partitioned into Σ_s and Σ_p, and we consider p-matches of a pattern to the text. Idury and Schäffer presented an AC-automaton like solution which occupies $\mathcal{O}(m \log m) = \mathcal{O}(n \log n)$ bits, and reports all *occ* occurrences in $\mathcal{O}(|T| \log \sigma + occ)$ time. Our main focus lies on the dynamic version of this problem. Specifically, we present the following results.

▶ **Theorem 1.** *By maintaining a linear-space index occupying $\mathcal{O}(n \log n)$ bits, we can answer:*

- $\mathsf{search}(T)$ *in worst-case $\mathcal{O}(|T| \log^2 n + occ)$ time.*
- $\mathsf{insert}(P_i)$ *in amortized $\mathcal{O}(|P_i| \log n)$ time.*
- $\mathsf{delete}(P_i)$ *in amortized $\mathcal{O}(|P_i| \log^2 n)$ time.*

▶ **Theorem 2.** *By maintaining a succinct-space index occupying* $(1+o(1))n\log\sigma+\mathcal{O}(d\log n)$ *bits, we can answer:*

- $\mathsf{search}(T)$ *in worst-case* $\mathcal{O}(|T|\log^2 n+occ)$ *time.*
- $\mathsf{insert}(P_i)$ *in amortized* $\mathcal{O}(|P_i|\log n)$ *time.*
- $\mathsf{delete}(P_i)$ *in amortized* $\mathcal{O}(|P_i|\log\sigma+\log d)$ *time.*

1.1 Roadmap

We show that if the patterns are appropriately encoded [5], then the problem can be solved using a generalized suffix tree GST of all the encoded patterns. Although the techniques are similar to that of Amir et al. [3] and Hon et al. [18], we need much more machinery to deal with parameterized patterns. This is because a crucial property, known as *suffix links*, of traditional suffix trees does not apply directly for parameterized strings. This makes navigating in the GST more tricky, and we have to augment the tree with additional data structures. Furthermore, it is difficult to maintain the analogous version of suffix links in the GST explicitly as they are more fragile to the deletion of patterns. Hence, we need an implicit representation. Also, following suffix links in the GST is trickier as the text and patterns have to be re-encoded. Moreover, maintaining the encoded patterns explicitly causes the space to increase to $n\log n$ bits as opposed to the $n\log\sigma$ bits occupied by the patterns.

The succinct solution is largely based on the sparsification technique [17, 18] for the (dynamic) dictionary matching problem. Broadly speaking, for a parameter Δ, the idea is to sample suffixes at an interval of Δ, and then maintain a GST for these sampled suffixes. Likewise, the text is also sampled. Now the sampled text starting from $i=1$ is matched, and all occurrences are reported. The occurrences reported in this run lie in the set $\{i, i+\Delta, i+2\Delta, \dots\}$. All occurrences are subsequently reported by repeating the process for $i=1,2,3,\dots,\Delta$. The sparsification technique, however, does not immediately extend to the case of parameterized matching. For one, handling and maintaining suffix links is trickier. Another issue is how to handle truncating of characters at the beginning of a currently matched text, which is essential for the approaches in [3, 17, 18].

In Section 2, we first present a linear space index and prove Theorem 1. This index forms the backbone of the succinct index (Theorem 2); the details are provided in Section 3. Section 4 discusses results on the semi-dynamic variant of the problem.

2 Linear Space Index

We assume that the alphabet Σ is disjoint from the set of integers. Any string S over Σ can be initially processed in $\mathcal{O}(|S|\log\sigma)$ time to ensure that this condition holds.

2.1 Parameterized Suffix Tree

Baker [5] introduced the following encoding scheme to enable matching of parameterized strings. Given a string S, obtain a string $\mathsf{prev}(S)$ by replacing the first occurrence of every p-character in S by 0, and any other occurrence by the difference in position from its previous occurrence. Thus, $\mathsf{prev}(AxByAxCz) = A0B0A4C0$, where $\{A,B,C\}\in\Sigma_s$ and $\{x,y,z\}\in\Sigma_p$. It is easy to see that $\mathsf{prev}(S)$ can be computed in $\mathcal{O}(|S|\log\sigma)$ time.[1] Baker showed that two strings S and S' are a p-match iff $\mathsf{prev}(S)=\mathsf{prev}(S')$. They introduced

[1] Read S from left to right, and use a balanced binary search tree (BST) to maintain the position of the latest occurrence of each p-character.

the *Parameterized Suffix Tree* (PST) of a string S, which is a compacted trie of the strings $\mathsf{prev}(S[i,|S|])$, $1 \leq i \leq |S|$. At each node u in PST, maintain $\mathsf{strDepth}(u)$ i.e., the length of the string formed by concatenating the edge labels from root to u. The label of an edge $e = (u, v)$ is derived dynamically as follows. We maintain two pointers from e to the start position sp and end position ep of the label in S. (Note that the encoding of the edge is not necessarily $\mathsf{prev}(S)[sp, ep]$.) Suppose we want to find the encoding of the jth character on e, where $\mathsf{strDepth}(u) = D$. Then we find the encoding $x = \mathsf{prev}(S)[sp + j - 1]$ using the pointers. If x is an s-character, then x is itself the desired encoding. Otherwise, if $x \geq D + j$ then the encoding is 0, else it is x. If $\mathsf{prev}(S)$ has been pre-computed and stored explicitly, then all operations require constant time per character. Suppose S_u is the string obtained by concatenating the labels (over S) of the edges from root to a node u. Then, the suffix link of u points to the location in the PST which is represented by $\mathsf{prev}(S_u[2, |S_u|])$. If $|S_u| \leq 1$, then the suffix link points at the root. Baker showed that unlike in suffix trees, a suffix link in PST can point to inside an edge.

Although Baker's encoding makes p-matching easier to handle, for our purposes, it suffers from a drawback. Specifically, $\mathsf{prev}(S)$ is a string over an alphabet of size $\Theta(n)$ in the worst case, whereas the original alphabet size σ may be much smaller in comparison. In order to alleviate this, our objective is to maintain S in $|S| \log \sigma$ bits so that we can still use the PST. In the above discussion, note that in order to find the prev-encoding of a p-character at the jth position, it suffices to find the last position (if any) in the interval $[sp - D, sp + j - 2]$ where the character $S[sp + j - 1]$ occurs. To facilitate this, instead of maintaining S explicitly, we build a Wavelet Tree [15] over it. Using this, we can easily find the desired encoding as follows (see Fact 3). Let $x = \mathsf{rank}(sp + j - 1, \mathsf{access}(sp + j - 1))$. If $x = 1$, the required encoding is 0. Otherwise, let $y = \mathsf{select}(x - 1, \mathsf{access}(sp + j - 1))$. If $y \geq sp - D$ then the encoding is $(sp + j - 1 - y)$ and is 0, otherwise.

▶ **Fact 3** ([15]). *Let S be a string of length m over an alphabet Σ of size σ. We can build a data structure in $\mathcal{O}(m \log \sigma)$ time that occupies $m \log \sigma + o(m \log \sigma)$ bits and supports the following operations in $\mathcal{O}(\log \sigma)$ time:*

- $\mathsf{access}(i) = S[i]$.
- $\mathsf{rank}(i, c) =$ *the number of occurrences of* $c \in \Sigma$ *in the substring* $S[1, i]$.
- $\mathsf{select}(j, c) =$ *the smallest position* i *such that* $\mathsf{rank}(i, c) = j$.

Suppose we are trying to find all p-matches of a string S' with S using the PST of S. Given a node v, in $\mathcal{O}(1)$ time, we can find the correct outgoing edge of v that matches the next (encoded) character of $\mathsf{prev}(S')$ by using a perfect hash function at each node. Specifically, the hash function maps the (encoded) first character of an edge to the edge itself. Every other p-character on an edge can be appropriately encoded as described above in $\mathcal{O}(\log \sigma)$ time. Therefore, the time to find all *occ* p-matches is $\mathcal{O}(|S'| \log \sigma + occ)$.

2.2 The Index

We assume that no two patterns P_i and P_j exist such that $\mathsf{prev}(P_i) = \mathsf{prev}(P_j)$. Recall that maintaining the patterns in their prev-encoded form requires $\Theta(n \log n)$ bits in the worst case. Although this will not affect our overall space (for the linear space index), we will use the following scheme, which would be carried forward to our succinct index. For every pattern $P_i \in \mathcal{P}$, we maintain a wavelet tree WT over P_i. Then we create a generalized parameterized suffix tree GST out of all the prev-encoded suffixes of $P_i\$_i$ and $P_i\#_i$, where $\$_i$ and $\#_i$ are two special s-characters neither of which belongs to Σ_s. Note that each leaf corresponds to the prev-encoded suffix of $P_i\$_i$ or $P_i\#_i$ for some pattern P_i. We maintain a link from

the leaf corresponding to the string $\mathsf{prev}(P_i[j, |P_i|])\$_i$ to the leaf corresponding to the string $\mathsf{prev}(P_i[j+1, |P_i|])\$_i$. Likewise, for the leaf corresponding to $\mathsf{prev}(P_i[j, |P_i|])\#_i$. This will help us in recognizing the suffix link of a node v implicitly.

For any node u, with slight abuse of notation, denote by $\mathsf{prev}(u)$ the string obtained by concatenating the encoded edge labels from root to u. As described in Section 2.1, (i) at each node u in the GST we maintain $\mathsf{strDepth}(u) = |\mathsf{prev}(u)|$ explicitly, and (ii) each edge is labeled by two pointers to the start and end positions of its label in a particular pattern. Using these and the WTs over the patterns, we can find the desired encoding of any character on an edge in $\mathcal{O}(\log \sigma)$ time (see Section 2.1). Furthermore, at each node we use the *Dynamic Perfect Hashing* technique of Dietzfelbinger et al. [11] such that given the next (encoded) character of the text, we can navigate to the appropriate edge (if any) in constant time. Moreover, we can update (both insert and delete) the hash table in amortized $\mathcal{O}(1)$ time. The total space needed to maintain the hash tables over all nodes is $\mathcal{O}(n \log n)$ bits.

Using the data structure of Sadakane and Navarro [27], we maintain a dynamic succinct representation of the GST (see Fact 4). The weight (for the purpose of wla queries) of a node u is $\mathsf{strDepth}(u) \leq n$. Clearly, the GST satisfies the min-heap property.

▶ **Fact 4** ([27]). *Given a dynamic tree with m weighted nodes, where a node's weight is greater than that of its parent. By encoding the tree topology in $2m + o(m)$ bits, we can support the following operations in $\mathcal{O}(\log m)$ time:*

- *Inserting or deleting a node.*
- *Lowest common ancestor* (LCA) *of two nodes.*
- *For any node, find its* (i) *pre-order rank,* (ii) *node-depth,* (iii) *parent,* (iv) *number of children,* (v) *ith leftmost child,* (vi) *number of sibling to its left,* (vii) *number of leaves in its subtree, and* (viii) *ith leftmost leaf in its subtree.*
- $\mathsf{levelAncestor}(v, D)$ *i.e., the node on the root to v path having node-depth D.*

Using this, in $\mathcal{O}(\log^2 m)$ time, we can find $\mathsf{wla}(u, W)$ i.e., the lowest ancestor (if any) of a node u that has weight at most W. This is facilitated by $\mathcal{O}(\log m)$ binary searches on the node-weights using levelAncestor *queries.*

For each pattern P_i, we locate the node u (which necessarily exists) such that $\mathsf{prev}(u) = \mathsf{prev}(P_i)$. We mark all such nodes with the corresponding pattern, and process the GST with dynamic nearest marked ancestor queries (see Fact 5). Consider a tree with m nodes, k of which are marked. Hon et al. [18] argued that by maintaining the order-maintenance data structure of Dietz and Sleator [10], the relative pre-order rank of two nodes can be compared in $\mathcal{O}(1)$ time. Furthermore, a node can be inserted into the data structure in $\mathcal{O}(1)$ time given either the predecessor or the successor (in pre-order) of the node; the node can also be deleted in $\mathcal{O}(1)$ time. Hon et al. used this to maintain the marked nodes in an interval tree. A marked node v is an ancestor of a node u iff the pre-order rank of u lies in the interval $[r_v, r_{v'}]$, where r_v and $r_{v'}$ are the pre-order ranks of v and of the last visited node in the subtree of v. The desired location where a new interval has to be inserted (or an existing one has to be deleted), can found in $\mathcal{O}(\log k)$ time using the interval tree. Likewise, the smallest interval which contains a node can be found in $\mathcal{O}(\log k)$ time; all subsequent intervals that encloses this smallest interval can be found in $\mathcal{O}(1)$ time per interval. The intervals in this tree are "elastic" in the sense that the pre-order ranks are compared in $\mathcal{O}(1)$ time "on the fly" using the order-maintenance data structure. Note that the pre-order rank of a node's successor/predecessor is found using Fact 4 in $\mathcal{O}(\log m)$ time.[2] Summarizing,

[2] The predecessor of each node is defined apart from the root. Given a non-root node u, its predecessor is

▶ **Fact 5** ([10, 18]). *Given a dynamic tree with m nodes and $k \leq m$ marked nodes. We can build an $\mathcal{O}(m \log m)$-bit data structure to support the following operations:*

- *Inserting or deleting a marked node in $\mathcal{O}(\log m)$ time.*
- *Report the K marked ancestors (if any) of a node in $\mathcal{O}(\log k + K)$ time.* ◀

2.3 Reporting Occurrences

A pattern P_i occurs at a position j in the text iff $\mathsf{prev}(P_i)$ is a prefix of $\mathsf{prev}(T[j, |T|])$. To find all patterns (if any) occurring at position j, we first find the deepest node v (called *locus*) such that $\mathsf{prev}(v)$ is a prefix of $\mathsf{prev}(T[j, |T|])$. Starting from v, we report all K marked ancestors of v using Fact 5 in $\mathcal{O}(\log d + K)$ time.

The task, therefore, is to find the locus of $\mathsf{prev}(T[j, |T|])$ for every $j \in [1, |T|]$ starting with $j = 1$. First we compute $\mathsf{prev}(T)$ in $\mathcal{O}(|T| \log \sigma)$ time. We use the WT and the edge pointers to traverse the GST starting from the root as follows. If we are at a node x, we use $\mathsf{prev}(T)[\mathsf{strDepth}(x) + 1]$ to select the correct edge in $\mathcal{O}(1)$ time. If we are inside an edge, then we use the next character of edge, say c, and verify it with the next character of $\mathsf{prev}(T)$. If c is static then it is easy. Otherwise, c needs to be encoded (as in Section 2.1) requiring $\mathcal{O}(\log \sigma)$ time. We continue this process until we hit a position $(k + 1)$ in the text such that the (encoded) character does not match. Let the corresponding edge be (u, v), where u is the locus of $\mathsf{prev}(T)$. Now, we need to find the locus of $\mathsf{prev}(T[j, |T|])$, where $j = 2$. We differ from the strategy of Amir et al. [3] in that we follow the suffix link of v instead of u.[3] (If $\mathsf{strDepth}(u) = k$, then follow the suffix link of u.) Recall that we do not explicitly maintain suffix links (other than in leaves). The following two cases are to be considered.

- $\boldsymbol{T[j-1] = T[1]}$ **is an s-character**: In this case, the suffix link necessarily points to a node w and $\mathsf{prev}\big(T[j, |T|]\big) = \mathsf{prev}\big(T[j-1, |T|]\big)[j, |T|]$. Our task is to locate the prefix of $\mathsf{prev}(w)$ which equals $\mathsf{prev}(T[j, k])$ (in this case, $j = 2$). Note that this prefix necessarily exists. We first locate a leaf ℓ in the subtree of v. Follow the pointer from ℓ to the leaf ℓ' depicting the starting position of the immediate next suffix as that of ℓ. We use the query $\mathsf{wla}(\ell', k - j + 1)$ to locate a node w'. If $\mathsf{strDepth}(w') = k - j + 1$, then we are done. Otherwise, we use the character $\mathsf{prev}(T[j, |T|])[\mathsf{strDepth}(w')]$ to select the proper edge. The desired location is given by $(k - j + 1 - \mathsf{strDepth}(w'))$ on this edge. Finally, we start matching from $T[k + 1]$ as defined previously until we hit a mismatch, resulting in the desired locus of $\mathsf{prev}(T[2, |T|])$.
- $\boldsymbol{T[j-1] = T[1]}$ **is a p-character**: The suffix link of v may point to the middle of an edge, say (x, y). Also, in this case as the encoding of T has to be modified. Specifically, $\mathsf{prev}(T[j, |T|])$ and $\mathsf{prev}(T[j-1, |T|])[j, |T|]$ may no longer be the same. Observe that for any j', if $\mathsf{prev}(T)[j']$ points to a location before j, then the desired encoding at j' is 0. Thus, we can easily update the encoding in $\mathcal{O}(1)$ time as characters are read. The correct position to start matching from $T[k + 1]$ can be found as described in the previous case by initially choosing a leaf in the subtree of v.

Summarizing, every time we locate the locus of $\mathsf{prev}(T[j, |T|])$, we truncate the character $T[j]$ by following the suffix link, obtain the encoding of $\mathsf{prev}(T[j + 1, |T|])$ if required, and

---

its parent v if u is the leftmost child of v; otherwise, the predecessor is the rightmost leaf in the subtree of its immediate left sibling. For the root node, its successor is its leftmost child.

[3] This is because if the first character of the current suffix (in this case, $T[1]$) is parameterized, then the suffix link from u can point to the middle of an edge (u', v'). Suppose, after reading the next characters we found a mismatch on the edge itself. Taking the suffix link from u' will push back us further and we may end up comparing too many characters.

then use the next characters of the text to find the locus of $\mathsf{prev}(T[j+1,|T|])$. By repeating the process, we will have located the locus of $\mathsf{prev}(T[j,|T|])$ for every $j \in [1,|T|]$.

The space occupied by the index is clearly $\mathcal{O}(n \log n)$ bits. Choosing the correct outgoing edge (if any) at any node takes $\mathcal{O}(1)$ time. Finding the leaf for an implicit suffix link operation takes $\mathcal{O}(\log n)$ time. Each weighted level ancestor query takes $\mathcal{O}(\log^2 n)$ time and WT query takes $\mathcal{O}(\log \sigma)$ time. Therefore, the time to find the loci is $\mathcal{O}(|T| \log^2 n)$, and the total time to report all occurrences is $\mathcal{O}(|T| \log^2 n + occ)$.

2.4 Handling Updates

We assume the pattern P_i that is to be inserted is not present in the dictionary. Likewise, for deletion, the pattern is present in the dictionary. Both can be easily verified in $\mathcal{O}(|P_i| \log \sigma)$ time by traversing the GST.

Insertion: To modify the GST, we use the algorithm of Kosaraju [21] which constructs the parameterized suffix tree PST of a string S in $\mathcal{O}(|S| \log \sigma)$ time. The algorithm, an adaptation of the McCreight's construction algorithm [24] for the traditional suffix tree, creates the PST by successively inserting the suffixes at positions $1, 2, \ldots, |S|$. Suffix links in the case of PST may point to the middle of an edge. These are termed as *bad* suffix links while the others (pointing to a node) are termed as *good* suffix links. Contrary to McCreight's algorithm, it no longer holds that every node other than the last entered leaf and its parent have good suffix links defined. For a node v, if $\mathsf{prev}(v)$ starts with an s-character then the suffix link of v is necessarily good. This allows insertion of suffixes starting with s-characters to remain the same as in case of McCreight's algorithm. Baker [5] showed that bad nodes (i.e., nodes with bad suffix links) have an outgoing edge labeled by a 0 and also form a chain in the PST. The number of bad nodes in this chain is at most $|\Sigma_p|$. Baker used this crucial observation to locate the desired bad suffix link to be followed for entering the next suffix, culminating in an $\mathcal{O}(|S|(|\Sigma_p| + \log \sigma))$ construction algorithm.

Kosaraju showed that by maintaining two pointers *low* and *high* to the lowest and highest nodes in the chain, the construction algorithm of Baker can be improved to $\mathcal{O}(|S| \log \sigma)$ when Σ_s is a constant-sized alphabet. Basically, the low and high pointers allow us to binary search on the chain of bad nodes to locate the proper position, rather than searching the entire chain. This improves the $|\Sigma_p|$ term to $\log |\Sigma_p|$. Kosaraju first created two separate suffix trees: (i) $\mathcal{T}_1$ for S with all p-characters replaced by 0 and (ii) $\mathcal{T}_2$ for S with all s-characters replaced by a single s-character. The first tree $\mathcal{T}_1$ can be constructed using Baker's algorithm and $\mathcal{T}_2$ using Kosaraju's algorithm for the constant-sized static alphabet. Using these trees, the final suffix tree is created. The trees are pre-processed with the data structure in [7] to support constant time LCA and $\mathsf{strDepth}$ queries for efficiently finding longest common prefix (LCP) information. For each suffix insertion, the number of such queries is $\mathcal{O}(\log |\Sigma_p|)$.

We show how to update the index for inserting a pattern P_i using the techniques above. The location to insert the first suffix i.e., $\mathsf{prev}(P_i)$ can be found by traversing the GST in $\mathcal{O}(|P_i| \log \sigma)$ time. Each suffix insertion in the GST will incur a cost of $\mathcal{O}(\log \sigma)$ for the $\mathcal{O}(\log |\Sigma_p|)$ number of LCA queries in $\mathcal{T}_1$ and $\mathcal{T}_2$, and $\mathcal{O}(\log n)$ time for inserting a constant number of nodes in the dynamic representation of the GST. Whenever a new node is to be inserted in the GST, we update the hash table in amortized $\mathcal{O}(1)$ time. The data structure of Fact 5 is modified once (insertion of a marked node corresponding to P_i in GST) and requires $\mathcal{O}(\log n)$ time. Finally, when the GST is constructed we will maintain the good suffix links (constructed by Kosaraju's algorithm) for each leaf corresponding to each suffix

of P_i. The WT for P_i can be constructed in $\mathcal{O}(|P_i| \log \sigma)$ time (see Fact 3). Thus, a pattern P_i can be inserted into the index in amortized $\mathcal{O}(|P_i| \log n)$ time.

Deletion: First, we find the locus u of $\mathsf{prev}(P_i)$ and unmark u. The time required is $\mathcal{O}(|P_i| \log \sigma + \log n)$. Then, we locate the loci of $\mathsf{prev}(P_i[j, |P_i|])$, $1 < j \leq |P_i|$. Let u be any such locus. Note that there are two edges out of u labeled by $\$_i$ and $\#_i$. Delete these edges and the corresponding children of u. There are two cases to be considered.

- ***u* is a leaf:** Remove u and its edge to its parent v. If v has more than one child, then modify the hash table at v. Otherwise, v is a node with a single child x. Let y be the parent of v. Add an edge from y to x with the label as the concatenated label of the edges from y to v and v to x (achieved by assigning the edge pointers appropriately). Modify the hash table at y. Remove the node v along with its edges.[4]
- ***u* is an internal node:** Modify u by treating it as node v in the above case.

Recall that the edge labels are maintained via two pointers to the start and end positions in a particular pattern. Upon pattern deletion, we may still have existing edges in the GST which have pointers to the deleted pattern P_i. (This happens as P_i may share a common prev-encoded prefix with many other patterns.) Relabeling of these edges is achieved as follows. Each edge can be found while locating the loci of each prev-encoded suffix of P_i. Consider such an edge (x, y). After deletion, we find a leaf in the subtree of y which is labeled with a pattern $P_{i'}$ and the starting position j' of the particular suffix. Then the pointers of the edge are modified easily in $\mathcal{O}(1)$ time using $P_{i'}$, j', $\mathsf{strDepth}(x)$, and $\mathsf{strDepth}(y)$.

Locating the loci requires $\mathcal{O}(|P_i| \log^2 n)$ time. For each locus, we perform a constant number of operations, each requiring amortized $\mathcal{O}(\log n)$ time (for modifying the data structure of Fact 4 and the hash table). Also, we relabel each edge correctly in $\mathcal{O}(\log n)$ time. The number of such edges is bounded by $\mathcal{O}(|P_i|)$. Finally, the WT corresponding to P_i can be easily deleted in $\mathcal{O}(1)$ time. Thus, the total time is bounded by $\mathcal{O}(|P_i| \log^2 n)$.

3 Succinct Index

We maintain a WT over each pattern. This occupies $n \log \sigma + o(n \log \sigma)$ bits (refer to Fact 3). We design our index by classifying the patterns into *long* and *short* based on a parameter Δ to be defined later. For short patterns (having length less than Δ), we create a compacted trie and use a rather brute-force approach. On the other hand, reporting the occurrences of long patterns (having length at least Δ) requires more sophisticated techniques. The set of occurrences of long patterns and short patterns are mutually disjoint, and are handled separately as indicated in the following lemmas.

▶ **Lemma 6.** *Let $\mathcal{P}$ be a dictionary consisting of d long patterns. By maintaining each pattern in a WT and a data structure occupying $\mathcal{O}(\frac{n}{\Delta} \log n)$ bits, we can report all occ_ℓ occurrences in $\mathcal{O}(|T|(\Delta \log \sigma + \log^2 n) + occ_\ell)$ time. Also, a long pattern P_i can be inserted in amortized $\mathcal{O}(\frac{|P_i|}{\Delta}(\Delta \log n + \log^2 n))$ time and deleted in amortized $\mathcal{O}(\frac{|P_i|}{\Delta}(\Delta \log \sigma + \log^2 n))$ time.*

▶ **Lemma 7.** *Let $\mathcal{P}$ be a dictionary consisting of d short patterns. By maintaining each pattern in a WT and a data structure occupying $\mathcal{O}(d \log n)$ bits, we can report all occ_s*

[4] Observe that there might still be a suffix link from a node v' pointing to the position corresponding to v on this new edge because truncating the first character of $\mathsf{prev}(v')$ may lead to merging of two outgoing edges of v'. Our motivation for implicit representation of suffix links is due to this property of PST.

occurrences in $\mathcal{O}(|T|(\Delta \log \sigma + \log d) + occ_s)$ *time. Also, a short pattern* P_i *can be inserted or deleted, both in amortized* $\mathcal{O}(|P_i| \log \sigma + \log d)$ *time.*

Theorem 2 is immediate by choosing $\Delta = \lceil \log n \log_\sigma n \rceil$, where $\epsilon > 0$ is an arbitrarily small constant. We proceed to prove the above two lemmas.

In what follows, we will assume the total length n of the patterns remains reasonably stable. This assumption is natural as we can use the following strategy of Overmars [28], or its subsequent improvement by Mäkinen and Navarro [23]. Roughly speaking, apart from maintaining the wavelet trees over the patterns, we will maintain three copies of the remaining component of the data structures in Lemmas 6 and 7. Specifically, apart from the data structures due to the choice of Δ above, we will keep two more copies, one for $\Delta = \Delta_{-1}$, and the other for $\Delta = \Delta_1$, where $\Delta_k = \lceil \log(2^k n) \log_\sigma (2^k n) \rceil$. Whenever the total length of the pattern doubles, we discard the structure for Δ_{-1}, and start building another structure by considering $\Delta = \Delta_2$. Likewise, when the total length halves, we discard the structure for Δ_1, and start building a structure by considering $\Delta = \Delta_{-2}$. Amortized per operation cost is $\mathcal{O}(1)$. Whenever, a pattern is inserted or deleted, we will modify all three copies simultaneously; a search query can be answered using any one of the copies. Clearly, the space-and-time bounds claimed in Lemmas 6 and 7 are not affected.

3.1 Long Patterns (Proof of Lemma 6)

For a string S and Δ, we use $\mathsf{head}(S)$ to denote the largest prefix of S whose length is a multiple of Δ and $\mathsf{tail}(S)$ is the remaining (possibly empty) suffix of S. We begin by obtaining $\mathsf{prev}(\mathsf{head}(P_i))$ for every $P_i \in \mathcal{P}$. We encode $\mathsf{tail}(P_i)$ from left to right using the same encoding that was used for $\mathsf{head}(P_i)$. More specifically, the desired encoding of the jth character in the tail is given by $\mathsf{prev}(P_i)[|\mathsf{head}(P_i)| + j]$. Then two equal-length strings S and S' are a p-match iff (i) $\mathsf{prev}(\mathsf{head}(S)) = \mathsf{prev}(\mathsf{head}(S'))$, and (ii) the encoded tails (as described here) of both S and S' are equal.

The Index: Note that in this case the number of patterns $d \leq n/\Delta$. We begin by sampling suffixes of each pattern head with sampling factor Δ. Specifically, for each pattern P_i, we obtain $\mathsf{prev}(P_i[k, |\mathsf{head}(P_i)|])$ for the suffixes starting at $k = 1, 1 + \Delta, 1 + 2\Delta, \ldots$. Starting from left, we group every Δ characters of these encoded suffixes. Let Σ' be an alphabet such that each character in Σ' corresponds to such a Δ-length substring. Replace the Δ-length substring by the corresponding character from Σ'. Create a generalized suffix tree $\mathcal{T}_{head}$ for all these suffixes of all the patterns. (If the pattern length is not a multiple of Δ, then we ignore its tail.) As in Section 2, we will append each condensed suffix with the special characters $\$_i$ and $\#_i$. Note that $\mathcal{T}_{head}$ has $\mathcal{O}(n/\Delta)$ nodes. Therefore, $\sum_u \delta(u) = \mathcal{O}(n/\Delta)$, where $\delta(u)$ is the number of outgoing edges of a node u. At each node u, we maintain $\mathsf{strDepth}(u)$, which is necessarily a multiple of Δ. Also, for each leaf ℓ, we maintain the pointers which will be used to find suffix links implicitly. The total space required is $\mathcal{O}((n/\Delta) \log n)$ bits.

Now, let us concentrate on how to navigate to a particular child of a node u. Consider all the outgoing edges of u. Create a compacted trie $\mathcal{T}_{head}(u)$ by treating the labels of these edges mapped to their corresponding Δ-length string. Note that each leaf in $\mathcal{T}_{head}(u)$ corresponds to a child of u in $\mathcal{T}_{head}$. Also, each edge in $\mathcal{T}_{head}(u)$ is labeled by a prev-encoded substring of a pattern P_i, and each outgoing edge of a node begins with a unique character from such a substring. As in the case of the linear space index, (i) at each edge of $\mathcal{T}_{head}(u)$ maintain the start and end pointers, and (ii) at each node maintain a dynamic perfect hashtable for navigating to the correct child based on the first (encoded) character of the edge. Since

the number of nodes in $\mathcal{T}_{head}(u)$ is at most $2\delta(u)$, the total space needed to maintain this information over all nodes in $\mathcal{T}_{head}$ is $\mathcal{O}(\sum_u \delta(u) \log n) = \mathcal{O}((n/\Delta) \log n)$ bits.

Now, we focus on the tail of each pattern. Consider a pattern P_i. First, we obtain the encoded tail of P_i (as described in the beginning of this section). Create two copies of the resultant tail, each of which is obtained by appending the s-characters $\$_i$ and $\#_i$. Locate the (distinct) node u in $\mathcal{T}_{head}$ such that $\mathsf{prev}(u)$ is same as $\mathsf{prev}(\mathsf{head}(P_i))$. Note that u is defined, and we call it the *head-node* of P_i. Consider all patterns which have the same head-node u. Create a compacted trie for the encoded tails of all those patterns, and let u be the root of that trie. We call this the *tail-trie* of u, and denote it by $\mathcal{T}_{tail}(u)$. The parent of each leaf in $\mathcal{T}_{tail}(u)$ corresponds to a pattern, say P_j, in the dictionary. We mark all such nodes in $\mathcal{T}_{tail}(u)$, and label them with the corresponding pattern index j. If there is a pattern P_j with an empty tail, then the corresponding tail-trie contains the head-node u, which is marked, and two leaves labeled by $\$_j$ and $\#_j$. The space occupied by each node for marking and labeling is $\mathcal{O}(\log n)$ bits. Each edge in $\mathcal{T}_{tail}(u)$ is labeled by a substring (of length less than Δ) of the encoded tail of a pattern. As in case of head tries, we maintain the two pointers on the edge to the corresponding pattern, and a perfect dynamic hash table to navigate to the correct child based on the first (encoded) character of the edge. This occupies $\mathcal{O}(\log n)$ bits for each node and edge. Since there are d patterns, the number of nodes and edges in all tail-tries combined is $\mathcal{O}(d)$. Since $d \leq n/\Delta$, the total space occupied for maintaining all tail-tries is $\mathcal{O}((n/\Delta) \log n)$ bits.

Denote the resultant trie by $\mathcal{T}_{long}$. We pre-process the head-trie with the data structure of Fact 4. Likewise, each tail-trie is pre-processed with the data structures of Facts 4 and 5. In summary, the total space occupied by $\mathcal{T}_{long}$ is $\mathcal{O}((n/\Delta) \log n)$ bits.

Reporting Occurrences: Starting from the position $j = 1$, we obtain $\mathsf{prev}(T[j, |T|])$ in $\mathcal{O}(|T| \log \sigma)$ time. Use it to traverse the trie $\mathcal{T}_{long}$ from the root. Each p-character labeling the edge of $\mathcal{T}_{long}$ can be properly encoded in $\mathcal{O}(\log \sigma)$ time as described in Section 2.1. Suppose, we have traversed up to node u in $\mathcal{T}_{head}$ and the character j' in $\mathsf{prev}(T)[j, |T|]$, where $j' - j + 1 = 0 \mod \Delta$. If $\mathcal{T}_{tail}(u)$ is not empty, then use the less than Δ characters of $\mathsf{prev}(T[j, |T|)$ starting from $j' + 1$ to traverse the tail trie, until we find a mismatch or reach a leaf. The time required is $\mathcal{O}(\Delta \log \sigma)$. Now, we use the marked ancestor data structure to report all occurrences starting at j corresponding to those patterns having head-node u. The time required is $\mathcal{O}(\log d + occ_{j,u})$ time. After this, by using the first Δ-characters of $\mathsf{prev}(T[j, |T|)$ starting from $j' + 1$, we have to select an edge (u, v) in $\mathcal{T}_{head}$. This is easily achieved in $\mathcal{O}(\Delta \log \sigma)$ time by using the navigation trie $\mathcal{T}_{head}(u)$ as follows. If we are at a node in $\mathcal{T}_{head}(u)$, then use the next character to find the correct edge using the hash table; otherwise, simply use the edge pointers to encode the next character of the edge, and match it with the next encoded character of T. In case we were no longer able to reach a leaf in $\mathcal{T}_{head}(u)$, then we have the following two scenarios. If no match was found with the first Δ characters starting from u, then we take the suffix link of u. Otherwise, we are necessarily on an edge to a leaf in $\mathcal{T}_{head}(u)$; in this case, take the suffix link of the node v in $\mathcal{T}_{head}$ corresponding to this leaf. In either case, we truncate Δ characters of $\mathsf{prev}(T)$ starting from j. As described in Section 2, the suffix link is simulated by the implicit suffix link i.e., by finding a leaf under u or v in the head-trie, and then using the leaf pointer and a wla query. Following this, the correct position to start matching is located in $\mathcal{O}(\Delta \log \sigma)$ time using the navigation trie of the node returned by the wla query. As before, locating a leaf requires $\mathcal{O}(\log n)$ time and a wla query takes $\mathcal{O}(\log^2 n)$ time. The number of times we have to select a proper edge, traverse a tail trie, or follow a suffix link, are all bounded by $\mathcal{O}(|T|/\Delta)$.

At the end of this process, for $j = 1$, we have reported occurrences of all patterns which start at a position of the form $j, j + \Delta, j + 2\Delta, \ldots$. The time required to find the loci and traversing the tail tries is $\mathcal{O}(|T| \log \sigma + \frac{|T|}{\Delta}(\Delta \log \sigma + \log^2 n))$. The time required to report the occurrences is $\mathcal{O}(\frac{|T|}{\Delta} \log d + occ_j)$. By repeating with $j = 2, 3, \ldots, \Delta$, all occ_ℓ occurrences of long patterns are reported in $\mathcal{O}(|T|(\Delta \log \sigma + \log^2 n) + occ_\ell)$ time.

Handling Updates: First we construct the head-trie when a pattern P_i is inserted. We begin by using Kosaraju's algorithm to construct a PST for P_i, and then find the locus of $\mathsf{prev}(P_i)$ in $\mathcal{T}_{head}$; this will take $\mathcal{O}(|P_i| \log \sigma)$ time. Now, we will create *actual* nodes in $\mathcal{T}_{head}$ only for those suffixes which start at a location of the form $k = 1, 1 + \Delta, 1 + 2\Delta, \ldots$. For other suffixes, we will create *dummy* nodes in $\mathcal{T}_{head}$ so as to perform the suffix link operations correctly. Specifically, suppose we have inserted an actual leaf ℓ_j for the suffix starting at $1 + j\Delta$. Subsequently, we will construct dummy leaves for the suffixes starting at $j' \in [2 + j\Delta, (j + 1)\Delta]$. Once, the actual leaf ℓ_{j+1} for the suffix starting $1 + (j + 1)\Delta$ is inserted, we will add the suffix link from ℓ_j to ℓ_{j+1}, and delete the intermediate dummy nodes. However, now we need to find the correct location of a (possibly new) node u in $\mathcal{T}_{head}$ such that $\mathsf{prev}(u)$ is the LCP of the suffixes corresponding to ℓ_j and ℓ_{j+1}, which is divisible by Δ. This can be found in $\mathcal{O}(\log^2 n)$ time using wla-queries on $\mathcal{T}_{head}$ by first finding the LCP using the PST of P_i. Each actual node insertion will take $\mathcal{O}(\Delta \log \sigma)$ amortized time for updating the structure of the navigation trie and the associated hash table, $\mathcal{O}(\log^2 n)$ time for wla-queries, and $\mathcal{O}(\log n)$ time for updating the data structure of Fact 4; the number of these operations is $\mathcal{O}(|P_i|/\Delta)$. We will make $\mathcal{O}(|P_i|)$ accesses for updating and querying the data structure of Fact 4 for inserting and deleting dummy nodes, each requiring $\mathcal{O}(\log n)$ time. Thus, the time needed to update $\mathcal{T}_{head}$ is $\mathcal{O}(|P_i| \log n + \frac{|P_i|}{\Delta} \log^2 n)$.

Modifying the tail-trie is much simpler. We traverse it with the encoded $\mathsf{tail}(P_i)$ starting from the head node of P_i until no more traversal is possible. Then, simply add the desired nodes and edges. Modify the data structures of Facts 4 and 5 accordingly (the latter for including a new marked node). Also, modify the hash table in $\mathcal{O}(1)$ amortized time per update. The time required is $\mathcal{O}(\Delta \log \sigma + \log d)$.

Since $|P_i| \geq \Delta$, inserting P_i needs amortized $\mathcal{O}(|P_i| \log n + \frac{|P_i|}{\Delta} \log^2 n)$ time.

In case of deletion, first we find the head-node of the pattern P_i. Then, use the encoded $\mathsf{tail}(P_i)$ to traverse the tail trie, unmark the node labeled by P_i, and delete its two children (leaves) labeled with $\$_i$ and $\#_i$. Also, the parent u of these leaves are deleted in case u is a leaf. The parent v of u is modified (if it has a single child) as in the case of linear index. If u is not a leaf, then it is treated in the same way if it has a single child, or else is left unmodified. To modify the edge pointers, find the lowest edge e that was traversed, but was not deleted. Then all the desired edges above e on the traversed path can be renamed by using any leaf corresponding to a pattern $P_{i'}$ under e. The hash table entries are deleted accordingly. The time required is $\mathcal{O}(|P_i| \log \sigma + \log d)$.

Deletion in the head trie is achieved by first locating the loci of all the condensed heads in time $\mathcal{O}(\frac{|P_i|}{\Delta}(\Delta \log \sigma + \log^2 n))$. Then, modify the edge labels in the navigation trie, and the adjoining hash table entries. Also, collapse nodes with a single child into an edge. The number of such operations is $\mathcal{O}(\frac{|P_i|}{\Delta})$, each requiring $\mathcal{O}(\Delta \log \sigma + \log n)$ time.

Since $|P_i| \geq \Delta$, deleting P_i needs amortized $\mathcal{O}(\frac{|P_i|}{\Delta}(\Delta \log \sigma + \log^2 n))$ time.

3.2 Short Patterns (Proof of Lemma 7)

Processing short patterns is similar to that for tail-tries. We create a compacted trie $\mathcal{T}_{short}$ for the strings $\mathsf{prev}(P_i)\$_i$ and $\mathsf{prev}(P_i)\#_i$. As in case of tail tries, we maintain the two pointers

for each edge, and also maintain the first (encoded) character of the edge in a dynamic perfect hashtable. Mark a node u if there is a pattern P_i such that $\mathsf{prev}(v) = \mathsf{prev}(P_i)$. Finally, we process the trie with the data structures of Facts 4 and 5. Since the number of patterns is at most d, the number of nodes in the trie is $\mathcal{O}(d)$, and the total space is $\mathcal{O}(d \log n)$ bits.

To find the occurrences of short patterns, first obtain $\mathsf{prev}(T)$ in $\mathcal{O}(|T| \log \sigma)$ time. Starting from $j = 1$, use $\mathsf{prev}(T)[j, \Delta - 1]$ to traverse the trie $\mathcal{T}_{short}$ until no more traversal is possible. The time required is $\mathcal{O}(\Delta \log \sigma)$. Now, starting the from the last encountered node, we report all occ_j occurrences starting at j in $\mathcal{O}(\log d + occ_j)$ time. We repeat the process for $j = 2, 3, \ldots, |T|$. The time required to report all occ_s occurrences is $\mathcal{O}(|T|(\Delta \log \sigma + \log d) + occ_s)$.

Insertion and deletion is similar as in the case of tail tries. Specifically, use $\mathsf{prev}(P_i)$ to traverse $\mathcal{T}_{short}$, and then insert/delete nodes accordingly. The hash table for navigation and the edge labels are also updated. Summarizing, both insertion and deletion needs amortized $\mathcal{O}(|P_i| \log \sigma + \log d)$ time.

4 Semi-Dynamic Dictionary

From the discussions in the previous section, closely observe that the $(\log^2 n)$-factor in the query complexity is due to the wla queries. To improve this, we present Fact 8.

▶ **Fact 8** ([20]). *Given a min-heap with m weighted nodes, with weights in $[1, m]$. We can build an $\mathcal{O}(m \log m)$-bit data structure in $\mathcal{O}(m)$ time to support the following operations.*

- insert *a weighted node maintaining the heap property in amortized $\mathcal{O}(\log \log m)$ time.*
- *report* $\mathsf{wla}(u, W)$ *in worst-case $\mathcal{O}(\log \log m)$ time.*

In conjunction with the techniques previously presented, for the semi-dynamic case, where only search and insert operations are supported, we obtain the following couple of corollaries to Theorems 1 and 2. The bound in Corollary 10 is attained by choosing $\Delta = \lceil \log^{\epsilon} n \log_{\sigma} n \rceil$ in Lemmas 6 and 7, where $\epsilon > 0$ is an arbitrarily small constant.

▶ **Corollary 9.** *By maintaining an $\mathcal{O}(n \log n)$-bit index, we can answer* $\mathsf{search}(T)$ *in $\mathcal{O}(|T| \log n + occ)$ time, and* $\mathsf{insert}(P_i)$ *in amortized $\mathcal{O}(|P_i| \log n)$ time.*

▶ **Corollary 10.** *By maintaining an $(1 + o(1))n \log \sigma + \mathcal{O}(d \log n)$-bit index, we can answer* $\mathsf{search}(T)$ *in $\mathcal{O}(|T| \log^{1+\epsilon} n + occ)$ time, and* $\mathsf{insert}(P_i)$ *in amortized $\mathcal{O}(|P_i| \log n)$ time.*

References

1. Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
2. Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *39th Annual Symposium on Foundations of Computer Science, FOCS'98, November 8-11, 1998, Palo Alto, California, USA*, pages 534–544, 1998. doi:10.1109/SFCS.1998.743504.
3. Amihood Amir, Martin Farach, Zvi Galil, Raffaele Giancarlo, and Kunsoo Park. Dynamic dictionary matching. *J. Comput. Syst. Sci.*, 49(2):208–222, 1994. doi:10.1016/S0022-0000(05)80047-9.
4. Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. La Poutré, and Alejandro A. Schäffer. Improved dynamic dictionary matching. *Inf. Comput.*, 119(2):258–282, 1995. doi:10.1006/inco.1995.1090.
5. Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 71–80, 1993. doi:10.1145/167088.167115.

6 Djamal Belazzougui. Succinct dictionary matching with no slowdown. In *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, pages 88–100, 2010. doi:10.1007/978-3-642-13509-5_9.

7 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, pages 88–94, 2000. doi:10.1007/10719839_9.

8 Sudip Biswas, Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Forbidden extension queries. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India*, pages 320–335, 2015. doi:10.4230/LIPIcs.FSTTCS.2015.320.

9 Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. Dynamic dictionary matching and compressed suffix trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 13–22, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070436.

10 Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 365–372, 1987. doi:10.1145/28395.28434.

11 Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994. doi:10.1137/S0097539791194094.

12 Guy Feigenblat, Ely Porat, and Ariel Shiftan. An improved query time for succinct dynamic dictionary matching. In *Combinatorial Pattern Matching – 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings*, pages 120–129, 2014. doi:10.1007/978-3-319-07566-2_13.

13 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.

14 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Succinct non-overlapping indexing. In *Combinatorial Pattern Matching – 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 – July 1, 2015, Proceedings*, pages 185–195, 2015. doi:10.1007/978-3-319-19929-0_16.

15 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 841–850, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

16 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 397–406, 2000. doi:10.1145/335305.335351.

17 Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Compressed index for dictionary matching. In *2008 Data Compression Conference (DCC 2008), 25-27 March 2008, Snowbird, UT, USA*, pages 23–32, 2008. doi:10.1109/DCC.2008.62.

18 Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Succinct index for dynamic dictionary matching. In *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, pages 1034–1043, 2009. doi:10.1007/978-3-642-10631-6_104.

19 Ramana M. Idury and Alejandro A. Schäffer. Multiple matching of parameterized patterns. In *Combinatorial Pattern Matching, 5th Annual Symposium, CPM 94, Asilo-*

mar, California, USA, June 5-8, 1994, Proceedings, pages 226–239, 1994. doi:10.1007/3-540-58094-8_20.

20 Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 565–574, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283444.

21 S. Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees (preliminary version). In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 631–637, 1995. doi:10.1109/SFCS.1995.492664.

22 Moshe Lewenstein. Parameterized pattern matching. In *Encyclopedia of Algorithms*, 2015. doi:10.1007/978-3-642-27848-8_282-2.

23 Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3), 2008. doi:10.1145/1367064.1367072.

24 Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

25 J. Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V. Thankachan. Top- k term-proximity in succinct space. In *Algorithms and Computation – 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, pages 169–180, 2014. doi:10.1007/978-3-319-13075-0_14.

26 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007. doi:10.1145/1216370.1216372.

27 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014. doi:10.1145/2601073.

28 Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983. doi:10.1007/BFb0014927.

29 Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Algorithms and Computation, 11th International Conference, ISAAC 2000, Taipei, Taiwan, December 18-20, 2000, Proceedings*, pages 410–421, 2000. doi:10.1007/3-540-40996-3_35.

30 Dekel Tsur. Top-k document retrieval in optimal space. *Inf. Process. Lett.*, 113(12):440–443, 2013. doi:10.1016/j.ipl.2013.03.012.